NASA/TM—2002-211981

Engineering Analysis Using a Web-Based Protocol

James D. Schoeffler
Ohio Aerospace Institute, Brook Park, Ohio

Russell W. Claus
Glenn Research Center, Cleveland, Ohio

October 2002

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the Lead Center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA's counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results . . . even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA Access Help Desk at 301–621–0134

- Telephone the NASA Access Help Desk at 301–621–0390

- Write to:
  NASA Access Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076

NASA/TM—2002-211981

Engineering Analysis Using a Web-Based Protocol

James D. Schoeffler
Ohio Aerospace Institute, Brook Park, Ohio

Russell W. Claus
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center

October 2002

Available from

NASA Center for Aerospace Information
7121 Standard Drive
Hanover, MD 21076

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22100

Available electronically at http://gltrs.grc.nasa.gov

Engineering Analysis Using a Web-Based Protocol

James D. Schoeffler
Ohio Aerospace Institute
Brook Park, Ohio 44142

Russell W. Claus
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

1. Abstract

This paper reviews the development of a web-based framework for engineering analysis. A one-dimensional, high-speed analysis code called LAPIN was used in this study, but the approach can be generalized to any engineering analysis tool. The web-based framework enables users to store, retrieve and execute an engineering analysis from a standard web-browser. We review the encapsulation of the engineering data into the eXtensible Markup Language (XML) and various design considerations in the storage and retrieval of application data.

2. Introduction

Most organizations seek to design and develop new products in increasingly shorter time periods, ref. 1. At the same time, increased performance demands require a team-based multidisciplinary design process that may span several organizations, ref. 2. One approach to meeting these demands is to modify the traditional product design approach by enabling rapid transfer of design information among team-members using a combination of XML data transfers and the redesign of engineering applications. This paper explores one such approach. We review the modification of an engineering analysis simulation (Lapin) to allow execution over a network using web-based protocols. This design enables users to store, retrieve and execute engineering analyses from a standard web-browser.

Engineering analysis has traditionally involved the execution of Fortran codes by reading input file(s) that contain initialization data (e.g. boundary conditions) for the calculation. The results of the execution of these codes are subsequently stored in separate output files. Sharing the results of these analyses has been problematic for both the original engineer as well as the engineer's design team members. Initially, there had to exist some means by which the original engineer could disseminate the results with other design team members. The design team members may then have had to perform some manipulation of these results (i.e. "massaging") in order to translate the data into a form suitable for their own analyses. Failures in the two-step process of distribution and data "massage" can occur and lead to anachronistic states in which out of date results and design information were used in conjunction with the latest design analysis.

The timely sharing of information is a major strength of the World Wide Web (WWW). WWW protocols are designed to allow the presentation of information that can be continuously updated. However, the file-based execution of Fortran codes does not readily lend itself to this new approach. There is no clearly defined standard for input and output data, thereby, making it difficult to share information between differing computing platforms. The eXtensible Markup Language (XML) provides one mechanism to address these problems. XML allows any data to be encapsulated in a platform independent manner and provides a mechanism to develop a standard for input / output data that can be shared with others. This XML data can then be passed to a web server that executes the engineering analysis and provides an output XML stream to any user on the network. Furthermore, these input and output streams may be stored in a database, allowing both archival storage and retrieval of analysis data.

This paper reviews various design considerations for the conversion of engineering applications toward a WWW-based protocol. We explore various options in the XML encapsulation of input and output data. The design of data storage to provide high-performance storage and retrieval is reviewed. Finally, we demonstrate web protocols for data access and execution.

3. Lapin Application

Lapin is quasi-one-dimensional unsteady flow analysis for supersonic inlets, reference 3. It is a Fortran application that requires an input dataset defining inlet geometry, initial flow conditions and other numeric parameters to control the solution algorithm. These input parameters are read by Lapin during execution and an output file is created containing solution results. This process suggests a simple object structure as seen in figure 1.

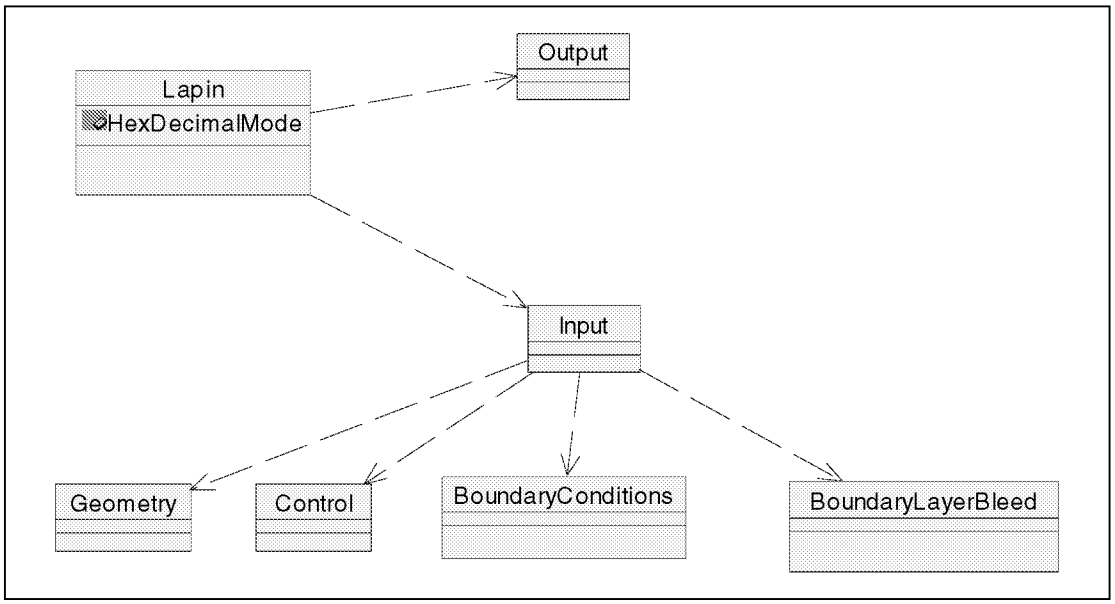

Figure 1. Overall Unified Modeling Language (UML) description of Lapin simulation data.

Lapin objects are organized hierarchically such that they do not contain closed paths of objects that reference one another. Many other aerospace applications objects exhibit a similar hierarchic structure, differing only in the number of sub-objects and depth of the sub-trees. The sub-objects associated with the Input object contain arrays of data. All input arrays have the same maximum size and all output arrays also have the same maximum size (but different from the input arrays). The total count of simple data members in these 7 classes is 63, and the total count of arrays is 62:

Class	Data Members	Arrays
Lapin	7	None
Input	None	None
Control	16	4
Geometry	13	31
BoundaryLayerBleed	3	12
BoundaryConditions	14	4
Output	6	11

Representative numbers for array sizes are 20 for the input arrays and 200 for the output arrays. The 62 data arrays represent a total of 1020 floats (51×20) for the short arrays and 2200 (11×200) for the long arrays or a total of 3220 floats.

4. Data Interchange Standard

Data standards are like a dictionary that provides a clear description of data semantics. In other words, it is important that an application understands what a term like "length" means and how it is described (e.g. units). For this report, we describe data standards that are solely useable by a Lapin application. Input and output data values are defined for use in the Lapin code. Of course, this is a demonstration that can be enlarged to allow integration of multiple codes as long as a consensus data standard is developed.

For this report, input and output information are captured in an XML representation. The representation is defined through a series of meta tags that describe any encapsulated data, ref. 4. The design pattern of these tags is captured in a Document Type Definition (DTD) or Extensible Stylesheet Language (XSL) document that can be used to test the validity of any XML dataset. A valid XML dataset must follow the structure defined in the DTD or XSL documents. Users with access to this specific DTD or XSL can parse a valid XML representation to extract some or all of "self-documented" data.

In this report, the parsing of the XML document is done by C++ custom-programmed applications that accessed the XML data using the Document Object Model (DOM), ref. 5, as is more fully discussed in Section 4a. This data was then passed to a C++ wrapper that encapsulates the FORTRAN Lapin code.

The design of meta tags used in XML typically follows an object structure such as the one illustrated in Figure 1. However, a variety of design decisions must be made to encapsulate the object structure within XML. Each of these decisions can affect the size of the resulting XML representation of a Lapin object and the size of the representation is important. Larger representations are undesirable because they increase storage needs and may increase data transfer times between a client and the server.

Various XML design decisions include:

1. Choice of the length of the name used to identify each object, or data member in the XML string.
2. Representation of a data member as an XML element or as an XML attribute.
3. Format for the value of a data member of type integer or float in the XML string.
4. Format of the individual values of an array of floats in the XML string.

Each of these design decisions in turn affects the length of the XML string representation. Furthermore, each decision can also affect the generality of the design in the sense of its applicability to other objects containing numerical data grouped into sub-objects.

For the Lapin analysis used in this study, the following design choices were made:

1. The number of data member names is small. Therefore, the name length has little impact on the overall XML string size, so the Lapin objects and the XML string use the same name. As a consequence, there is no need to invent new names and then map them to the actual C++ objects that are used to parse the XML . This implies that a name extracted from the XML string can be passed as a string to the function building the Lapin C++ object (along with the value) and hence the function becomes non-XML specific.

2. The use of attributes or elements for specification of data members has almost no affect on the overall XML string size. Attributes are not used because the XML schema would require an attribute name to be created for each attribute. For the same reason, elements specific to each data member are not used. The result is the use of a general element (with some name such as "DM") with the use of one attribute to contain the data member name. The type is not stored as an attribute implying that the Lapin function which creates the XML string or receives data from the XML string decoder must associate the name and type from knowledge of the data member of the actual objects. This must be done anyway for error-checking purposes. Should the need arise to store the type, an attribute can be added without making existing XML strings obsolete.

3. The XML string representing the application data is large, so there is a trade-off between the utility of a readily readable ascii representation and a smaller, unreadable data representation. The use of hex digits rather than decimal digits

decreases the string size very significantly and was adopted for all numeric variables (integers and floats). Attributes of the Lapin object are used to specify the format's numeric base, floating-point format, and byte order so that the string can be used to build Lapin objects in other computers. It is then possible to change numeric formats without adversely affecting much of the code for the creation or decoding of XML strings should the circumstances dictate the need for use of decimal digits.

A further consequence of this decision is that a text string is used for the value as far as XML itself is concerned as opposed to an XML type such as "r4" which knows that the value is an IEEE 801 format 4 byte float. Conversion then becomes the responsibility of the Lapin functions creating and using the values in the XML string.

4. Array values so dominate the XML string that all values were stored as fixed-length hex strings. Thus, an array becomes a data member with a "single" value which is a long string (e.g., 1600 characters for an array of 200 floats).

The Lapin object structure is shown in Figure 1 and the XML representation of this structure is described in the following items.

1. The single outer XML element is called "Lapin". This object has three attributes, which are global to all other XML elements. The attribute "HexDecimalMode" determines whether the integers and floats and arrays are specified in hex or decimal digits. Two other attributes, InputArraySize and OutputArraySize, define the length of arrays in the input and output objects.

2. An element named "Input" is a sub-element of "Lapin". It contains sub-elements named "class". Four such elements named "Geometry", "BoundaryConditions", "Control", and "BoundaryLayerBleed" are the sub-elements of "Input" object.

3. An element named "Output" is an optional sub-element of "Lapin" also. It is optional so that a Lapin object may be created before a simulation is run and there is no output data available. After the analysis has been run, the "Output" object can be added to a newly generated XML string.

4. Elements corresponding to the 4 types of data elements are declared in the schema. They are "DMI" for an integer data member, "DMB" for a boolean data member, "DMF" for a single float data member, and "DMFA" for an array of floats. Because an array is stored as a single string of hex or decimal digits, an array is no different from any simple data member except for the size of the representation. These data member elements are sub-elements of the various Lapin objects and sub-objects containing data members.

5. Unlike some languages (e.g. the Unified Modeling Language –UML), the XML specification does not permit the exact specification of number of sub-elements. For example, a Lapin object has exactly 4 sub-objects. The schema specifies that there are "1 or more" "Input" objects. This limitation of XML must be handled by the Lapin object code which generates the string (it must generate exactly 4 input objects) and the Lapin object code which decodes the XML string and generates the Lapin C++ object (it must check that there are exactly 4 "Input" elements with the proper names ("Geometry", etc.) before creating the Lapin C++ objects.

4a. Conversion of an XML string to a Lapin object -the parsing problem

The XML Document Object Model (DOM) was used to parse the XML representation to provide the data values needed by the Lapin analysis (i.e. the Lapin object). C++ code was used to both generate the XML representation of a Lapin object and for the parsing of the XML representation to recreate the Lapin object. The design of the schema assumed the specific Lapin class representation noted earlier. The Lapin C++ code was structured to allow efficient coordination between the DOM object parsing of the XML string and the use of the retrieved elements and attributes to create the Lapin object. A helper object that paralleled the Lapin object provided efficient data structuring. The XML helper object required a state-based control so that an element could be parsed in depth (including sub-objects) and then the state of the parser would recognize completion of that element and restore the state so that the next element at the higher level could be parsed. This allowed, for example, the parser to discover that the Lapin object contained sub-objects (Geometry, etc.), recreate those objects by interacting with the Lapin C++ classes, then continue parsing XML from the base Lapin object.

4b. Conversion of a Lapin object to an XML string

Each Lapin sub-object was designed to be able to generate XML elements and attributes corresponding to its data members. All were generated as DOM objects. DOM objects were connected in a tree that represented the structure of the Lapin object. The DOM itself simply generates the XML string representation of the tree in order to create the Lapin XML string. This is effective because of the extensive DOM error checking of the data and structure legality via the Lapin schema (as captured in XSL).

5. Data Storage and Retrieval Design

Each XML dataset is intended to represent a single Lapin calculation with the Input and Output data fields. This XML data can be stored in two different locations. On the server, a SQL database was used to store the Lapin XML representations. A client browser can query the database and perform a search based on user-defined characteristics, but the data is stored on the server. On the client, XML strings can be stored in an Excel spreadsheet as detailed in Section 5.3.

XML is used as the “linga franca” or common data language to store the Lapin simulation data. Each external application that uses this information must transform

between XML to the application-specific data format. XML is used as an intermediary to avoid the need for direct translations from one application to another. The direct translation approach would require N*N custom translators, whereas the use of XML as an intermediary reduces this need to 2N.

Experiments run from the design discussed in this paper found that parsing an XML string and creating the C++ Lapin object in memory takes approximately 125 milliseconds on a 300MHz Pentium. Creating an XML string from a C++ object in memory takes approximately 180 milliseconds.

5.1 SQL Database Storage Considerations

A database representation is important for two reasons: persistence of specific objects over a long time period and interactive examination and retrieval of objects. The examination and retrieval of objects requires a retrieval language that can search the database for objects satisfying specific criteria. Relational databases provide a standard language, Structured Query language (SQL), that allows retrieval based upon (possibly) complex boolean relationships involving the contents of as many data fields as desired. The downside of this is the need to store all the data fields of all sub-objects of the hierarchical object in tables in the database.

Representation of an object in a relational database is usually done by creating one table in the database for every different class of object and providing one field for each data member of that object-class. All instances of objects of that class then appear as a table with one row per object-instance and each column representing the values of a given data member of the class for each object instance. There are several complications however as discussed in the following sections.

5.1.1 Representation of object key data as fields in tables

If individual sub-objects objects of a given hierarchical object are stored in separate tables, it must be possible to determine for each object in each table the given base object it is part of. This requires that a unique identifier or key be created for each base object and stored as a field in each sub-object-table. To make the key readable by a user, a text key was used in the test implementation. The key identified the creator of the object, the date and time created, and a project name. Since the key must be stored in each table, it would be desirable to have a shorter key but then would not allow a user viewing a sub-object-table to associate each row with a specific base object.

5.1.2 Representation of array data as database "blobs"

A "blob" in database terms is simply an arbitrarily-sized data item whose internal structure and data values are not known to the database software itself. The name comes from the description "binary large objects". A blob is stored and retrieved as a unit and may not be involved in a query. That is, a query may not refer to a value in a Blob to be

used in selecting which items are retrieved. The blob element in a database table is actually a pointer to the arbitrarily-sized data somewhere else in the database.

It would be possible to create a "blob" for each array in the object associated with the table. The corresponding column name could be the name of the array in the object. This results in a simple database design whose tables contain all the data for a sub-object in one row. Alternatively, the database could be designed with a single "blob" containing all 62 arrays in one field of the base-object table. Either of these alternatives could be used for retrieving array data and building the corresponding Lapin objects.

Neither alternative allows the use of array data in database queries nor does it allow the user of the database to browse the database examining object data for the purpose of deciding which objects to retrieve. The power of a relational database lies in its SQL query capability. Hence these alternatives were not considered further since full query capability involving data contained in input and output arrays is necessary. However great advantage can be taken of a blob-field as discussed in the next section.

5.1.3 Representation of array data in array-size specific tables

Five of the six Lapin objects contain data arrays. SQL Server table columns must contain a single named value. This implies that storing that data in the table would be roughly equivalent to using separate data members for each element of each array. Since table columns must be named, this requires the creation of a unique name for each array element (e.g., the array name followed by its index). This approach results in tables with many columns which is not itself a problem. However using the artificial column names in a query makes queries involving these data items awkward to create.

Alternatively, each of the arrays would have to be treated as a separate object and stored in a separate table. This leads to excessive time to access the database while storing and retrieving an object.

An alternative appropriate to objects commonly used for aerospace simulations takes advantage of the assumption that all input arrays commonly have the same maximum size and all output arrays also have the same maximum size. The two sizes are quite different however. In this alternative, array data is stored separate from the objects that contain the data. Two additional tables are used, one for input data-arrays and one for output data-arrays. One row of a table represents one array. Each table must have a column which contains a key identifying the simulation object to which the array belongs and another column identifying the name of the array. These tables are quite manageable. For the common maximum sizes of 20 for input arrays and 200 for output arrays, the tables contain 22 and 202 columns respectively. Column names can then conveniently be the index value of the array element.

5.1.4 Embedding the XML string in a table

Since the fundamental representation of an object is an XML string, it is possible to avoid retrieving sub-objects provided that the XML string representing the object is itself stored in the base object table. This was done in this representation for the purpose of providing a much faster retrieval of objects from the database since only the single XML string field must be retrieved because the entire object including sub-objects and arrays can be created from it.

This approach allows queries to be as complex and complete as desired, but retrieval remains simple because only one (long) string-field is retrieved. The alternative, retrieving the data members of each object from its corresponding table, and then creating the object in memory by storing the retrieved data values into the hierarchy of objects would do the same thing. However, the XML string would then have to be generated from the newly created object in memory. The retrieval of many fields from many tables, finding the rows associated with a given object, and constructing the object that way is a much more complex software task that significantly lengthens retrieval-time.

In practice, not all sub-objects are usually involved in the queries. In this case, it is not actually necessary even to store those non-involved sub-objects in the database since the XML string in the base object table is sufficient for complete recovery of the object.

The implementation discussed in this paper required 1 table per object and two tables for the input and output arrays with the XML string stored in the base object table for a total of 8 tables. The 6 sub-object tables contain 1 row per Lapin object. The input data array contains 51 rows for each Lapin object and the output array 11 rows per Lapin object.

5.2 Performance of the transformations between XML strings and the relational database

Two experiments were created for the purpose of getting insight into the performance of the database portion of the transformation between representations.

1. Create a new empty database in which a user-selected number of objects are created and stored in the database. Each object is stored in a row of multiple tables, one per sub-object but with all arrays stored in one of two tables as discussed above. The experiment allowed the option of storing the input and output data vectors in order to determine the effect of the array-storage choice on the response time.

2. Open an existing database created in the first experiment, retrieve a subset of the stored data, and build the corresponding Lapin objects from the retrieved data. Standard SQL queries are used. The result of a query is an array of pointers to the retrieved simulation objects.

The cost of retrieving all the data for the sub-objects including the array data directly from the database varies from 2 to 4 seconds per object. This time includes retrieving the data from 6 tables plus separate queries to the array storage tables to retrieve each of the 62 arrays, a total of 68 separate SQL server queries per object. This is a very inefficient way to use a relational database.

Better design of the queries could significantly reduce the above retrieval times at the cost of more complex transformation software helpers. However, the use of the stored XML string to allow completely general queries with simple retrievals is so superior to the direct retrieval of sub-object data that it is not worth considering. Parsing is faster than querying.

The following table shows the storage and retrieval times for various number of Lapin objects where the result of the general retrieval query actually retrieves only the XML string. It does not include the time required to construct the Lapin object from the XML string (this time is discussed below).

Number of Simulation Objects	Storage Time (seconds)	Retrieval Time (seconds)	Retrieval Time Per Object (secs)
1	2.7	0.69	0.690
5	3.8	0.76	0.153
25	14.1	1.63	0.065
50	24.9	2.66	0.043
75	37.7	3.61	0.048
100	49.3	4.82	0.048

Notice that storage time for an object includes the time to store all the data members of the six objects and the XML string in six tables and to store the 62 arrays in the two array tables and is approximately 0.5 seconds per Lapin object when multiple objects are stored and around 1 to 3 seconds for a single object. The long times for storage of a single object is due to SQL server's caching of internal data in anticipation of further queries of a similar nature. Repeated single object queries then become faster.

Retrieval time varies from 69 milliseconds for 1 object to around 48 milliseconds per object when many objects are retrieved. Thus retrieval is generally 10 times faster than storage because of the retrieval of the XML string. Despite the size of the XML string (around 31,000 bytes in these examples) the fast retrieval time is well worth the cost.

If the data arrays did not have to be stored in the database for querying purposes, the storage time reduces considerably. A repeat of the above storage and retrieval experiments in the case data arrays (both input and output) are not stored in the data base yields the following results:

Number of Simulation Objects	Storage Time (seconds)	Retrieval Time (seconds)	Retrieval Time Per Object (secs)
1	1.52	0.62	0.615
5	1.74	1.16	0.233
25	2.63	1.74	0.070
50	5.49	2.85	0.057
75	7.76	3.95	0.053
100	9.41	4.55	0.046

Retrieval times are about the same as expected since only the XML string is actually retrieved. However storage times drop from the 2–4 second range to the 0.9–1.5 second range, a decrease of 2 times for a single object to a decrease of 5 times for multiple objects.

Since object creation from an XML string is approximately 125 milliseconds, the total time to retrieve an object from the SQL server database and construct the corresponding object in memory ranges from 170 milliseconds to 775 milliseconds per object.

5.3 Performance of the transformations between XML strings and Excel spreadsheets

The Excel spreadsheet representation includes a worksheet for each sub-object that contains the names and values of all data members as well as the array data members formatted in a readable fashion. In addition, the XML string itself is embedded into a hidden cell on the worksheet corresponding to the base object.

The transformation from an XML string to an Excel spreadsheet takes about 4 seconds to load the Excel program and 25 seconds to parse the XML string and write the comment and data cells into the spreadsheet. The actual workbook contains 6 separate sheets that must be created, with a total of 3754 separate entries that have to be written to cells. The elapsed time includes parsing the XML string, creating the pages, determining the location on the sheet to write the next cell, and actually writing the cell. This includes all data members and their names as comments, all arrays including their names and a column of index values, the embedded XML string in a hidden cell, and the creation of the 6 spreadsheets to hold the above data from the sub-objects. During the writing, the spreadsheet is set to the non-update mode so that it does not attempt to redraw itself each time a cell is changed.

The 25 seconds of elapsed time corresponds to an average write time per cell of 6.7 milliseconds for each of the 3754 cells. Considering that the program creating the spreadsheet is in a separate process from the Excel process and that an automation interface to Excel is being used, a multi-millisecond write time is usual. Note that this corresponds to a write rate of 150 cells per second. The 3754 cells actually consist of 120 cells corresponding to titles, descriptions, and data of simple data members of each of the

Lapin objects, 1210 cells for the input arrays' descriptions and data, and 2424 cells for the output arrays' descriptions and data. Hence the writing time of Lapin data without arrays takes about 1 second, the input arrays about 8 seconds, and the large output array data about 16 seconds. These times were calculated from the average cell write time and were not directly measured.

After a user change to the spreadsheet, the transformation from the spreadsheet to a new XML string takes 15 seconds. This is faster because cells are read but not written except for the embedded XML string that is changed to the new XML string representing the data actually present in the Excel spreadsheet. Only cells containing data (not descriptions) are read and these represent half of the 3754 cells. Note, that the elapsed update time for half of the cells is 20% more than half of the elapsed write time for all the cells. Hence, the Excel read time per cell is approximately the same as the write time per cell.

Currently the Excel spreadsheet program is opened and no sheet is displayed until all 6 sheets have been created and written. This leaves the user staring at a blank sheet. The apparent (but not actual) speed of the XML string to Excel spreadsheet representation could be improved by about 5 to 1 if the base object worksheet were written first and then displayed to the user while the other 5 sheets were being written in the background.

6. Web Protocols for Data Access and Execution

The process of encapsulating input and output data for a Lapin analysis calculation has been reviewed in the previous sections. This section discusses how these XML strings are used in a web-based environment. Merely capturing the input and output data in a structured format is a first step in this process. The next step involves designing the client/server interactions that clearly specify how these XML strings are used in the execution of the Lapin application.

Figure 2 displays a conceptual picture of the client/server approach used in this study. In a general sense, the client provides Lapin input data (in the form of XML strings) to the server. The server parses the input data and controls the execution of the Lapin application. Then the server provides an XML string that encapsulates the Lapin output data to the client. The exact details of the interaction are described in the following sections.

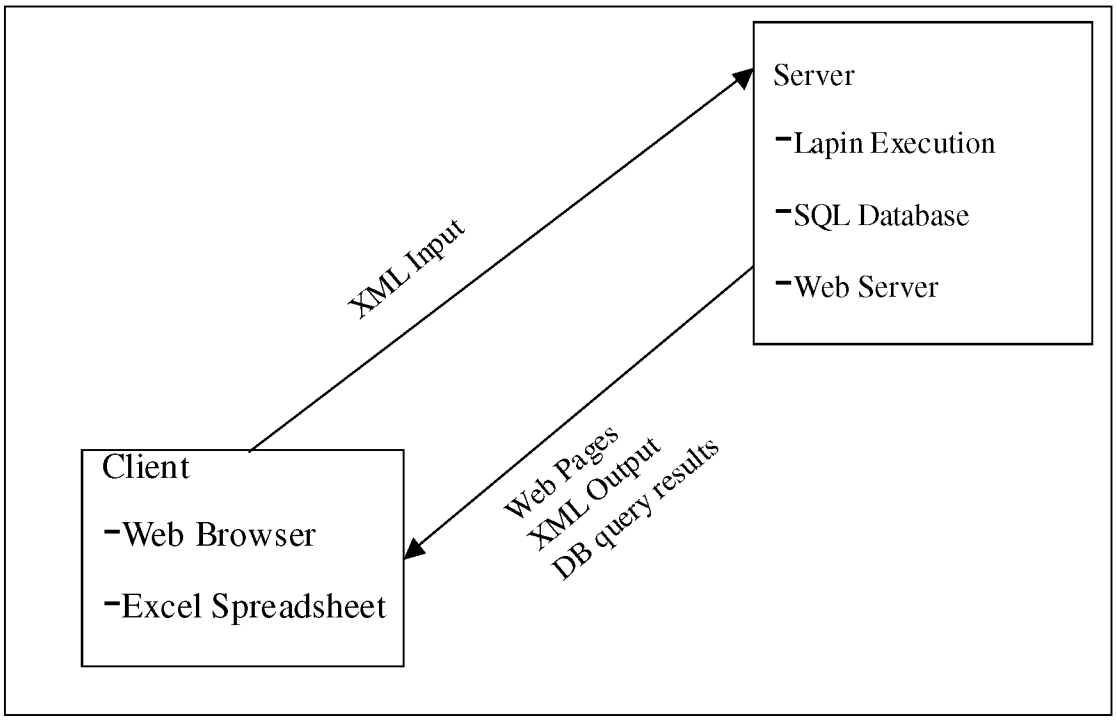

Figure 2. Client/Server interactions for the Lapin application.

6.1 The Lapin Web application and the design of its active-server pages

The interaction with a user of the application starts with the user navigating to the server site and the LapinWebApp.asp page using a standard browser (Internet Explorer or Netscape Navigator). The execution of this page causes an Hyper Text Markup Language (HTML) page to be sent to the user and displayed in the user's browser. The page displays a form showing the state of the application (e.g., whether or not a lapin object has been accessed) and the user's options (e.g., display a lapin object in an excel spreadsheet). This is illustrated in Figure 3. The response of the user causes either a local command or a remote command to be generated. A local command is one that is handled entirely on the client computer whereas a remote command is one that is transmitted to the remote server for handling.

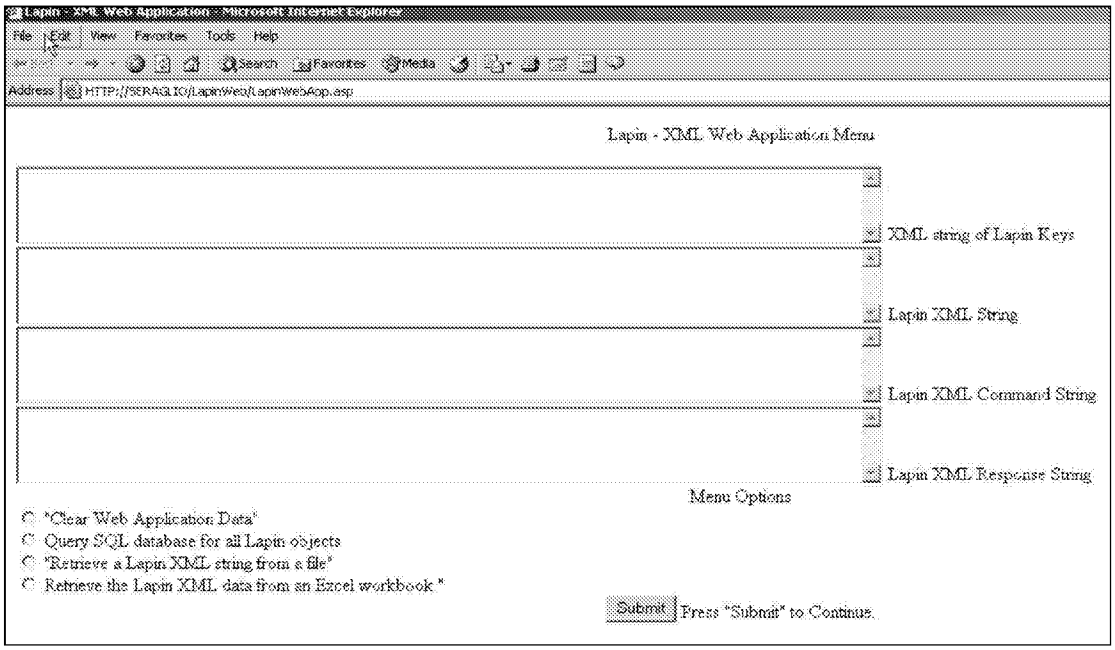

Figure 3. A view of the client browser screen upon initial contact with the server.

Local commands must be supported by application code on the user's machine. Examples in this application are the following:

1. Read an XML string encoding a lapin object from a local file.
2. Write the current lapin object XML string to a local file.
3. Display the current lapin object in an Excel spreadsheet.
4. Retrieve a lapin object from data in an Excel spreadsheet.

All of these local commands are most efficiently processed on the user's client workstation. The local lapin objects allow changes to values in a local spreadsheet that may be submitted for another run of the Lapin code.

Remote commands are supported at remote server machines (one or more) so that it is not necessary for a user to have complex applications installed on the user's local machine. Examples of remote applications are the following:

1. Read the keys of lapin objects stored in a database (SQL server). [As was described in Section 5.1.1, the key describes: the creator of the object, the date and time created, and a project name.]
2. Retrieve the lapin object from a key retrieved from the database.
3. Store the current lapin object into the SQL server database.
4. Transmit the current lapin object's XML string description to the server where the Lapin application code resides and run the analysis code.

The database commands are more naturally remote commands so that results of lapin object analysis runs can be centrally stored for retrieval and examination by many users.

Similarly, the analysis code often must reside on a secure server and therefore is not available to install on each user's machine.

In all cases of both local and remote commands, an HTML page is returned to the user with application state and options set to those currently meaningful. Figure 4 shows the returned HTML page after executing the remote command that accesses the SQL Server database and returns an XML string containing the keys of all lapin objects currently stored in the database. The user could select one of these keys and execute another remote command to return the corresponding lapin object in the form of an XML string.

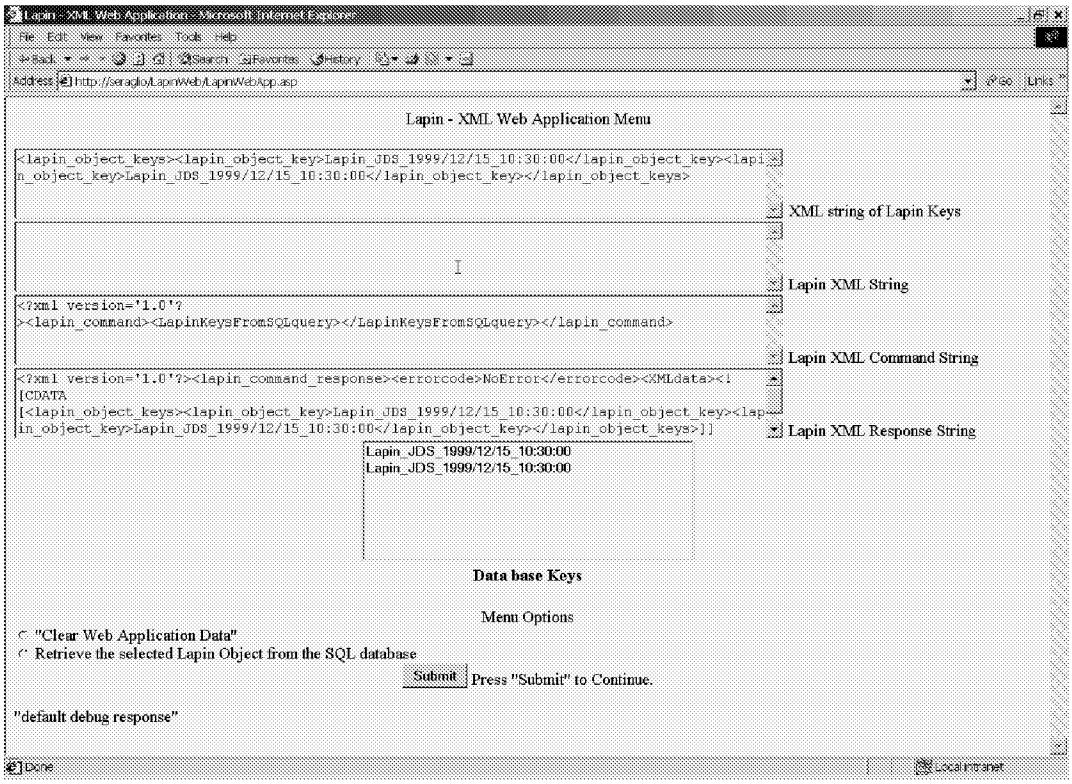

Figure 4 HTML page displaying typical results from a user request.

6.1.1 Handling of the User Interface "Submit" button

Figure 3 shows the appearance of the initial HTML page displayed so that the user may enter data and select a command for execution. After the user has entered whatever data is needed for a command (e.g., a file name and path) and selected the checkbox corresponding to the command to be processed (e.g., "retrieve XML from a file"), the following sequence of actions take place.

For each of the checkboxes displayed on the form in sequence

if the checkbox is checked (selected)

call the corresponding action subroutine
in the case of a local command, the action subroutine processes the command whereas in the case of a remote command, it calls ProcessXMLCommand to navigate to a remote ASP page to process the command.
write the hidden variable values to the form
the values of the variables are copied into the form variables that are in turn used in the generation of the returned ASP page.
actually submit the form for processing

Once submitted, an ASP response page is generated and displayed. The checkboxes are actually radio buttons so that only one may be clicked at any time. Clicking a second automatically unselects the first.

6.1.2 The round-trip from user data entry through receipt of the response of a command

The user interacts with a dialog displayed within the user's browser. The user has the option of entering data (e.g., a file name where a lapin object is located), browsing to the file location to pick from a list of files, and even editing information displayed in data-entry areas on the form. The user makes his/her choice of command from a set of radio-buttons each describing an available command, and clicks the 'submit' button to initiate the command.

The response to the submit button click determines whether the command is local (and if so carries it out locally). Local commands may use ActiveX controls but these must be available on the local machine. For example, file read and write controls are available through standard operating systems such as Windows and are used to read and write files of interest to the user on the local machine. Controls for generating and parsing XML strings also are standard under Windows. The implementation here used custom ActiveX controls for converting Lapin objects to XML data and for displaying Lapin objects in Excel spreadsheets and retrieving modified data from an Excel spreadsheet.

If the command is remote, the following sequence of actions take place.

1. An XML command string is generated that encodes the desired command and all the data needed to carry out the command. For example, the command to run the Lapin analysis program requires specification of input data. All of this data is stored in individual objects within a C++ Lapin object that is not suitable for transmission to a remote server.

Hence an XML string is generated from the Lapin object and this represents the input data. In the current application, this string is approximately 30,000 bytes in length and is a complete XML string. It is embedded into the command XML string as a single data item using the XML CData code. This causes an XML processor to read the entire string as data (and ignore its XML tags embedded in that data).

2. The XML command string is sent as data to the remote server where it is translated by an Active-Server Page (ASP), ref. 6.
3. A Visual Basic script parses the components of the command string. These components are the command name, and each parameter needed by the command. In the case of the command to run the Lapin analysis code, the name is "RunLapinF" and only a single parameter is supplied, the entire Lapin XML string. The XML string appears to be a single data item in the command string because of the use of the CDATA tag.
4. An ActiveX object appropriate to the particular command is created by Visual Basic scripts in the ASP page and a method (member function) of that object called passing the parameter data extracted from the command string.
5. The specific ActiveX object generally extracts the individual data items from the arguments passed to the function and prepares the data for the actual carrying out of the command. In the case of the RunLapinF command, the parameter is an XML string that encapsulates a large amount of data needed by the lapin analysis code. The XML string is processed by the XML processor, a Lapin object is created, and all the encapsulated data is used to initialize the lapin object and its 4 sub-objects. The C++ Lapin object is then used to move the data into the appropriate variables used by the Lapin analysis code. This is Fortran code and all of its input data is stored in a large block data statement.
6. The Lapin analysis code is started and it simply reads data from its block data variables and executes to produce output data also in variables in its block data area.
7. The output data is then packed into arrays and variables within the C++ Lapin output sub-object.
8. The updated lapin object then generates a new lapin XML string that contains not only the original input data, but also the newly generated output data.
9 A response XML string is then generated. It includes tagged error information and the CDATA-tagged updated Lapin XML string.
10. The response string is returned to the ASP page that generated the command and its components extracted.
11. If the error information component indicates a failure, an error message is generated to be returned to the user.
12. If the error information component indicates success, the Lapin XML string is extracted and stored in a variable to be returned to the user.
13. An HTML page updated according to the new state and including the command response returned to the user.

6.2 Key aspect of the design: maintenance of the state of the application

Web applications pass HTML and ASP web pages back and forth between the user at the local machine and the server machines where remote applications are run. Web applications are intrinsically state-independent in that there is no automatic maintenance of global data from one page to another. However it is almost always necessary to pass data from the one page to the other and to return data from a remote machine. In addition to data, state information must be passed along with data specific to a command and also returned with the command response. This permits maintenance of the state of the application between interactions. For example, initially, there is no current lapin object so the user's options are few. Later on, there may be a current lapin object and hence the user's options are different. In this design, a set of variables that determine the state of the application are maintained on the HTML pages as values of form variables that are transmitted to the server, updated there, and returned on the new HTML page sent back to the user. These form variables are invisible to the user but used by the page to determine how to proceed. For example, the HTML page returned to the user from a remote server contains script (commands) that check the values of the hidden variables to determine exactly what information should be displayed to the user for the current state of the application. The result is that the user sees a state-dependent display that changes after each command selected by the user.

6.3 The XML Command specification

The command sent from the user to a remote server has the following specification:

```
<XML version="1.0">
    <LapinCommand>
        <CommandName>name</CommandName>
        <Parameter1> data </Parameter1>
        .....
        <Parametern> data </Parametern>
    </LapinCommand>
```

For example, the command to store a Lapin object into the SQL database must pass the XML string as a parameter. The command is:

```
"<lapin_command>"
    "<XMLstringToDB>"
        "<lapin_xml_string>"
            "<![CDATA[" & LapinXMLString & "]]>"  _
        "</lapin_xml_string>"
    "</XMLstringToDB>"
"</lapin_command>"
```

The parameter, 'lapin_xml_string' is actually a string of length 30K bytes. It is treated as the value of a single tag by enclosing it in the CDATA form shown. Only one parameter is used with this command.

Once the server ASP page is activated, it extracts the XML command string, parses it and checks that it is a command by looking for the "LapinCommand" tag. It then extracts the value of the "CommandName" tag and processes the command using a Visual Basic script specific to the particular command. That script understands the number and types of parameter data that must be present, extracts those parameters, error checks the parameter data, and then creates the appropriate ActiveX control and passes the data to the control by calling a method on its interface. The ActiveX control returns command-specific data through the output variables of the interface methods or error data in case of failure of the command.

The following script shows how the data received from the user is processed and the XML parsed and checked:

```
'Create XMLDOM object, and load XML data from client
  Set docReceived = CreateObject("Microsoft.XMLDOM")
  docReceived.load(Request)

  set RootNode = docReceived.documentElement
  ' RootNode is the root element of a tree
  '     containing the XML received in the form of a DOM tree
  ' Test for the kind of tree (node type) and use that
  ' and the XML tree to get the data to be handled.
  receivedXML = docReceived.xml
 Dim RootNodeName  'the name of the message
 RootNodeName = RootNode.nodeName
 ' test for message type from client
If NOT RootNodeName="lapin_command" then
    ReturnXMLerror RootNodeName , "illegal lapin command"
    End If
```

The lapin command is processed differently for each supported command. The command to store a Lapin object into the SQL database is processed as follows:

```
elseif commandNodeName = "XMLstringToDB" then
      'parameter1 is the xml-string
      Set parameter1 =
            commandNode.selectSingleNode("//lapin_xml_string")
      if (err.number<>0) then
            ReturnXMLerror "601"," Command 'XMLstringToDB' is
                  missing parameter 'lapin_xml_string' "
           end if
     parameter1Value = parameter1.firstChild.text

     xml_result = LinkXMLobj.LapinXMLtoSQL( parameter1Value)
     if (err.number <> 0) then
            ReturnXMLerror "620", " Command 'XMLstringToDB' failed
                  for key '" & parameter1Value & "' " & err.number & " " &
                  err.description
    else
       ReturnXMLsuccess(xml_result)
```

Note that the ActiveX object used to handle this command is 'LinkXMLobj' and the particular method called is 'LapinXMLtoSQL. This example shows how the data returned is changed into an XML string by scripting functions.

The script that called the ActiveX control's method understands the arguments returned but must then change that data into a response XML string. The scripting functions are:

```
Sub ReturnXMLerror(ErrorLabelString, errorMessage)
    On Error Resume Next
    Set LinkXMLobj = Nothing
    responseXML = "<?xml version='1.0'?><lapin_command_response>" &_
      "<errorcode>" & "<![CDATA[" & ErrorLabelString & "]]>" & "</errorcode>" &_
      "<errormessage>" & "<![CDATA[" & errorMessage & "]]>" & "</errormessage>" &_
      "</lapin_command_response>"

    if err.number <> 0 then
      Response.End
    end if
   Response.Write responseXML
   Response.End
End Sub


sub ReturnXMLsuccess(xml_result)
    On Error Resume Next
    Set LinkXMLobj = Nothing
    responseXML = "<?xml version='1.0'?>" & _
    "<lapin_command_response>" &_
    "<errorcode>NoError</errorcode>" &_
    "<XMLdata>" & "<![CDATA[" & xml_result & "]]>" & "</XMLdata>" &_
    "</lapin_command_response>"

    if err.number <> 0 then
      Response.End
    end if
    Response.Write responseXML

    Response.End
    Exit Sub
End Sub
```

Notice that the return XML string is generated by simply concatenating strings containing the various XML tags and the data. Note that the success return function again returns the XML string of the Lapin object encapsulated in the CDATA statement.

6. 4 List of functions for client-side processing

The client ASP page (LapinWebApp.asp) uses several functions for error detection and utility purposes.

A function used for error tracing is: “ExtractLeadingXML.” This function searches for the XML tag ‘lapin_command_response’ and extracts an XML string from the CDATA

statement associated with this tag. This XML string must be embedded within a CDATA statement since an XML string is not a legal entity within another XML string. Both the response XML string (that comes from the server) and the embedded XML string can then be checked for legality.

"CheckXMLresponse" takes an XML response string and checks its XML format. It is used simply for error detection.

"CheckXMLparses" function does the actual XML legality check by making sure that the XML string may be parsed. A complete Document-Object-Model tree is created as part of the legality checking.

7. Summary

This paper has reviewed various design considerations for the conversion of an engineering application (Lapin) toward a WWW based protocol. The conversion of a Fortran code to allow web execution and data storage was found to be practical in concept and in practice. The use of XML for the encapsulation of input and output data was efficient and facilitated software development.

The design of data storage was critical to provide high-performance storage and retrieval. Data retrieval times could vary as much as a factor of five depending on how the data storage is configured. To achieve good data retrieval performance, only a few "high-level" or meta-data members were used for querying through the database. Array data values (the majority of the information) were stored in "blobs" that were keyed to the appropriate meta-data.

The use of a Lapin XML string as a common data source for all applications was efficient and minimized the number of custom data translators that were created. Experiments run from the design discussed in this paper found that parsing an XML string and creating the C++ Lapin object in memory takes approximately 125 milliseconds on a 300MHz Pentium computer.

8. Conclusion

Conversion of a Fortran-based engineering analysis (Lapin) into web-based analysis server has been demonstrated. Using XML to define application specific data allowed a unified representation that eased the process of transferring data between various applications. This was demonstrated by storage in a searchable SQL database and Excel spreadsheet. In the future, the XML encapsulated definitions could be used to allow so-called, "Smart" agents (or applications) to find needed information for a large scale simulations such as the Numerical Propulsion System Simulator, reference 7.

References

1. Smith, Preston, and Reinertsen, Donald G., "Developing Products in Half The Time," International Thomson Publishing, Inc., 1995.
2. Schrage, D.P., Gordon, M., "Management Issues and Techniques in Concurrent Engineering," AIAA 92–4206, August, 1992.
3. Varner, V.O, Martindale, W.R., Phares, W.J., Kneile, K.R., and Adams, J.C., "Large Perturbation Flow Field Analysis and Simulation For Supersonic Inlets," NASA CR–174676, September 1984.
4. Homer, A., "XML IE5", Wrox Press Ltd, 1999.
5. http://www.w3.org/DOM/
6. Powers, Shelley, "Developing ASP Components," O'Reilly and Associates, March 2001.
7. Claus, R.W., Evans, A.L., Lytle, J.K., and Nichols, L.D., "Numerical Propulsion System Simulation," Computing Systems in Engineering, Vol. 2, No. 4, pp. 357–364, 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 2002	Technical Memorandum

4. TITLE AND SUBTITLE

Engineering Analysis Using a Web-Based Protocol

5. FUNDING NUMBERS

WU–704–40–13–00

6. AUTHOR(S)

James D. Schoeffler and Russell W. Claus

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
John H. Glenn Research Center at Lewis Field
Cleveland, Ohio 44135–3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E–13651

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546–0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM—2002-211981

11. SUPPLEMENTARY NOTES

James D. Schoeffler, Ohio Aerospace Institute, 22800 Cedar Point Road, Brook Park, Ohio 44142; Russell W. Claus, NASA Glenn Research Center. Responsible person, Russell W. Claus, organization code 5880, 216–433–5869.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category: 61 Distribution: Nonstandard

Available electronically at http://gltrs.grc.nasa.gov

This publication is available from the NASA Center for AeroSpace Information, 301–621–0390.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

This paper reviews the development of a web-based framework for engineering analysis. A one-dimensional, high-speed analysis code called LAPIN was used in this study, but the approach can be generalized to any engineering analysis tool. The web-based framework enables users to store, retrieve, and execute an engineering analysis from a standard web-browser. We review the encapsulation of the engineering data into the eXtensible Markup Language (XML) and various design considerations in the storage and retrieval of application data.

14. SUBJECT TERMS

Software engineering; Object-oriented programming; Mechanical engineering

15. NUMBER OF PAGES

28

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	